Divide and Conquer Symmetric Tridiagonal Eigensolver for Multicore Architectures

Grégoire Pichon, Azzam Haidar, Mathieu Faverge, Jakub Kurzak

Divide and Conquer Symmetric Tridiagonal Eigensolver for Multicore Architectures

Grégoire Pichon*, Azzam Haidar†, Mathieu Faverge*, Jakub Kurzak†
*Bordeaux INP, Inria Bordeaux - Sud-Ouest, Talence, France
{gregoire.pichon, mathieu.faverge}@inria.fr
†Innovative Computing Laboratory
The University of Tennessee, Knoxville, USA
{haidar, kurzak}@icl.utk.edu

***Abstract*—Computing eigenpairs of a symmetric matrix is a problem arising in many industrial applications, including quantum physics and finite-elements computation for automobiles. A classical approach is to reduce the matrix to tridiagonal form before computing eigenpairs of the tridiagonal matrix. Then, a back-transformation allows one to obtain the final solution. Parallelism issues of the reduction stage have already been tackled in different shared-memory libraries. In this article, we focus on solving the tridiagonal eigenproblem, and we describe a novel implementation of the Divide and Conquer algorithm. The algorithm is expressed as a sequential task-flow, scheduled in an out-of-order fashion by a dynamic runtime which allows the programmer to play with tasks granularity. The resulting implementation is between two and five times faster than the equivalent routine from the INTEL MKL library, and outperforms the best MRRR implementation for many matrices.**

***Keywords*-Eigensolver, multicore, task-based programming, PLASMA, LAPACK**

I. INTRODUCTION

With the recent emergence of architectures with many cores, algorithms have to be re-designed to take advantage of the new hardware solutions. The level of parallelism to express becomes so significant that, according to Amdahl's law, some algorithms see their performance decrease dramatically due to the cost of their sequential portion. In addition, the fork/join model has reached its limits due to the cost of synchronizations on a number of resources that keep increasing. Recently, many runtimes have emerged to limit and find solutions to this problem. The task-based approach is a way to avoid many of those barriers as they can be expressed as local barriers, concerning a smaller number of tasks. A runtime schedules tasks on the multiple computational units available according to their dependency analysis. Thus, the sequential part of the original algorithm can sometimes be computed alongside other tasks. Furthermore, using tile algorithms allows us to take advantage of data locality by pipelining operations that exploit the same data. A contiguous memory can be divided up in the tile distribution in such a way that it provides a good task granularity depending on the architecture. Combining both approaches expresses more parallelism, and exploits current architectures as much as possible. Here, we focus on the symmetric eigenproblem for a multicore system, to see how such a task-based tile algorithm can outperform existing implementations.

The symmetric eigenproblem is a problem arising in many applications, such as quantum physics, chemistry and statistics. A standard approach for computing eigenpairs is first to reduce the matrix to a tridiagonal form T:

$$A = QTQ^T \tag{1}$$

where A is the considered dense n-by-n original matrix and Q an orthogonal matrix. Then, a tridiagonal eigensolver is used to compute eigenpairs of the tridiagonal matrix:

$$T = V\Lambda V^T \tag{2}$$

where V is the matrix of orthogonal eigenvectors, and Λ the diagonal matrix encoding the eigenvalues. Finally, the resulting eigenvectors are updated according to the original matrix:

$$A = (QV)\Lambda(QV)^T \tag{3}$$

The PLASMA project [1], [2] started a few years ago to revisit well-known algorithms of the LAPACK library, and to adapt their implementations to current multicore systems. PLASMA algorithms follow a task-flow model, which allows the programmer to utilize a huge level of parallelism through a fine task granularity. The symmetric eigenproblem has already been studied in PLASMA, with the implementation of a new routine to compute the tridiagonal reduction [3]. The back-transformation relies on matrix products and is already efficient on recent architectures. However, the solution of the tridiagonal eigenproblem is computed using LAPACK routines, and only exploits parallelism through level 3 BLAS operations if correctly linked with a multi-threaded implementation such as the INTEL MKL or the AMD ACML libraries. Thus, we would like to propose a new tridiagonal eigensolver, based on a task-flow programming model, on top of the internal QUARK runtime.

Regarding this problem, four algorithms are available: QR iterations, Bisection and Inverse Iteration (BI), Divide and Conquer (D&C), and Multiple Relatively Robust Representations (MRRR). The first two, QR and BI, are presented in [4], but a performance comparison made by Demmel et

al. [5] compared LAPACK algorithms and concluded that D&C and MRRR are the fastest available solvers. However, while D&C requires a larger extra workspace, MRRR is less accurate. Accuracy is a fundamental parameter, because the tridiagonal eigensolver is known to be the part of the overall symmetric eigensolver where accuracy can be lost. D&C is more robust than MRRR, which can fail to provide an accurate solution in some cases. In theory, MRRR is a $\Theta(n^2)$ algorithm, whereas D&C is between $\Theta(n^2)$ and $\Theta(n^3)$, depending on the matrix properties. In many real-life applications, D&C is often less than cubic while MRRR seems to be slower than expected due to the number of floating divisions and the cost of the iterative process. The main asset of MRRR is that a subset computation is possible, reducing the complexity to $\Theta(nk)$ for computing k eigenpairs. Such an option was not included within the classical D&C implementations, or it only reduced the cost of the updating phase of the last step of the algorithm [6]. Considering the fact that accuracy and robustness are important parameters, the objective of this paper is to introduce a new task-based Divide and Conquer algorithm for computing all eigenpairs of a symmetric tridiagonal matrix.

The rest of the paper is organized as follows. We present the sequential algorithm in Section III, followed by the parallelization strategy in Section IV. Then, Section V describes the testing environment with a set of matrices presenting different properties, and analyzes our results in comparison with existing concurrents. Finally, Section VI concludes with some prospects of the current work.

II. Related Work

As we mentioned in Section I, four algorithms are available in LAPACK [7] to solve the tridiagonal eigenproblem. In this part, we will focus only on D&C and MRRR, the fastest available solvers. However, different approaches can be used to solve the complete symmetric eigenproblem. The Jacobi eigenvalue algorithm [4] is an iterative process to compute eigenpairs of a real symmetric matrix, but it is not that efficient. Recently, the QDWH (QR-based dynamically weighted Halley) algorithm was developed by Nakatsukasa [8], and provides a fast solution to the full problem. Another approach is to reduce the matrix to band form (not especially tridiagonal form) before using a band eigensolver. It has been investigated in the past, but here we will focus more precisely on the tridiagonal eigensolver.

The Divide and Conquer algorithm is included in both LAPACK and ScaLAPACK [9], and – to our knowledge – no faster implementation is available for multicore architectures. The LAPACK version [10] is sequential, and exploits parallelism through level 3 BLAS operations. During the merge process, the main cost lies on two matrix products, and parallelizing those operations provides a good speedup with respect to the sequential execution. The first parallel implementation was proposed in [11] to exploit the idea of partitioning the original problem. ScaLAPACK [12] expands the previous implementation for distributed architectures. Contrary to LAPACK where subproblems are sequential, ScaLAPACK allows us to distribute the computation among all the available nodes. In addition, the merge steps present much more parallelism than the LAPACK version, where only the updating process performs BLAS 3 operations. Both LAPACK and ScaLAPACK implementations are included in the Intel MKL library [13].

MRRR, designed by Dhillon [14], has also been studied for multicore architectures. The objective of MRRR is to find a suitable standard symmetric indefinite decomposition LDL^T of the tridiagonal matrix, such as a small change in L causes a small change in D. Such a representation allows for computing eigenvectors with a relatively high accuracy. On the other hand, it requires well separated eigenvalues. The first representation divides the eigenvalues spectrum into subsets of close eigenvalues. Then, a new representation $L'D'L'^T = LDL^T - \sigma I$ is computed with σ chosen in order to break existing clusters. This process is repeated until each cluster contains only one eigenvalue. Then, eigenvectors can be computed. With such an approach, a good level of parallelism is expressed: an eigenvector computation is independent from the rest. However, the LAPACK version does not rely on level 3 BLAS operations, so it is more difficult to exploit parallelism. ScaLAPACK provides a parallel implementation for distributed architectures, but the available routine in the API is for the complete symmetric eigenproblem. The fastest implementation for shared-memory systems was developed by the Aachen Institute for Advanced Study in Computational Engineering Science in MR^3-SMP [15]. The algorithm is expressed like a flow of sequential tasks and relies on Posix threads. Their internal runtime can schedule tasks statically or dynamically. Their experiments showed how well it outperforms the original MRRR algorithm with naive fork/join parallelization. They also explain why the implementation is scalable, and how it exploits as much parallelism as possible. In addition, computational timing shows that MR^3-SMP is often better than the D&C from the Intel MKL library. It confirms that, as in many cases, the task-based approach is better suited than the fork/join model provided by the Intel MKL library for modern architectures. As Demmel et al. showed that D&C and MRRR are comparable in terms of performance for a sequential run, it motivates our development of a task-based D&C to see how well this approach can challenge MR^3-SMP.

A heterogeneous approach of D&C algorithm for the Singular Value Decomposition (SVD) problem has also been studied in [16]. The algorithm is close to the one used for the eigenproblem, and both the secular equation and the GEMMs are computed on GPUs. The speedup with respect to the Intel MKL library is interesting, but we cannot exploit those improvements in our task-based approach.

III. The sequential algorithm

A symmetric tridiagonal eigensolver computes the spectral decomposition of a tridiagonal matrix T such that:

$$T = V\Lambda V^T \text{ with } VV^T = I \tag{4}$$

where V are the eigenvectors, and Λ the eigenvalues. The original Divide and Conquer algorithm was designed by Cuppen [17] and later [18] proposed a stable version of this algorithm. This algorithm has been implemented by the state-of-art LAPACK and ScaLAPACK packages. We follow Cuppen's strategy in our design. The D&C approach can then be expressed in three phases:

- Partitioning the tridiagonal matrix T into p subproblems in a recursive manner forming a tree.
- The subproblems at the leaf of the tree are considered to be *simple* or *small* eigenvalue problems. Each of these problems may be considered as an independent problem without any data dependencies with the other leaves of the tree and solved by the classical QR iterations technique which provides an accurate solution.
- Merge the subproblems which are defined by a rank-one modification of a tridiagonal matrix, and proceed to the next level of the tree in a bottom-up fashion as shown in Figure 1.

Thus, the main computational part of the D&C algorithm is the merging process. In order to describe the merging phase and for the sake of simplicity, we define $p = 2$, but it is easy to generalize for any $p < n$, with n being the matrix size. Partitioning the problem with $p = 2$ gives:

$$T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \beta uu^T \tag{5}$$

where T_1 and T_2 are the two tridiagonal submatrices, with the first and the last element modified by subtracting β respectively, u is a vector where $u_i = 1$ only when $i = \frac{n}{2}$ or $i = \frac{n}{2} + 1$. Suppose that the solution of the eigenproblem of T_1 and T_2 are given by $T_1 = V_1 D_1 V_1^T$ and $T_2 = V_2 D_2 V_2^T$. Thus, the merge phase consists of solving the system:

$$T = \tilde{V}(D + \beta zz^T)\tilde{V}^T \tag{6}$$

where $\tilde{V} = diag(V_1, V_2)$ and $z = \tilde{V}^T u$. This system is solved by two steps, first we find the spectral decomposition of $R = D + \beta zz^T = X\Lambda X^T$ (which is called "*rank-one modification*") and then we explicitly construct the eigenvectors V of T by performing a matrix product of the updated eigenvectors X with the previous computed one's (the eigenvectors $\tilde{V}$ of the two sons) such as $V = \tilde{V} \times X$. Golub [19], [20] has shown that if the d_i (the components of D) are distinct and the ζ_i (the components of z) are different from zero for all i, then the eigenvalues/vectors of R are the zeros of $w(\lambda)$, where

$$w(\lambda) \equiv 1 + \beta \sum_{i=1}^{n} \frac{\zeta_i^2}{d_i - \lambda} \tag{7}$$

Equation 7 is known as the secular equation. Note that when the d_i are equal (meaning that D has multiple eigenvalues) or when $\zeta_i = 0$ or $|\zeta_i| = 1$ the problem is deflated meaning that some of its final eigenpairs (λ, v) are directly known without the need of any computation. As a result, the *deflation* process prevents the computation of eigenpairs (λ, v) of the merged system R, which are acceptable to be eigenpairs of the father node (here in this example, it is T). Therefore, it reduces both the number of secular equations and the size of the matrix product with $\tilde{V}$.

Finally, let us outline a detailed description of the merging phase. It proceeds in seven steps:

- finding the deflation;
- permuting the vectors $\tilde{V}$ in a way to have two sets (non-deflated and deflated vectors);
- solving the secular equations of the non-deflated portion (compute the updated eigenvalues Λ and the components of the updated eigenvectors X);
- computing a stabilization value for each eigenvector according to Gu technique (described in [18]);
- permuting back the deflated eigenvectors;
- computing the eigenvectors X of the updated system R;
- updating the eigenvectors of the father system $V = \tilde{V} \times X$.

The D&C approach is sequentially one of the fastest methods currently available if all eigenpairs are to be computed [4]. It also has attractive parallelization properties as shown in [12].

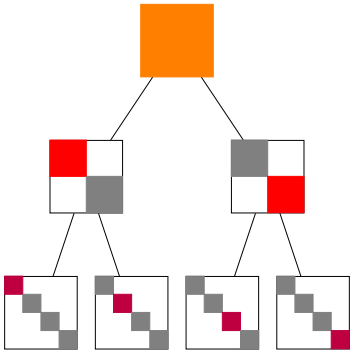


Figure 1. D&C merging tree

Given n the size of the problem and k the number of non-deflated eigenvalues, the cost of the different steps of the merging phase is listed in Table III.

In the worst case, when no eigenvalue is deflated, the overall complexity can be expressed by:

$$n^3 + 2(\frac{n}{2})^3 + 4(\frac{n}{4})^3 + \cdots = \sum_{i=0}^{log(n)} \frac{n^3}{2^{2i}} = \frac{4n^3}{3} + \Theta(n^2) \tag{8}$$

We can observe that the overall complexity is dominated by the cost of the last merge which is about n^3 operations.

Operation	Cost
Compute the number of deflated eigenvalues	$\Theta(n)$
Permute eigenvectors (copy)	$\Theta(n^2)$
Solve the secular equation	$\Theta(k^2)$
Compute stabilization values	$\Theta(k^2)$
Permute eigenvectors (copy-back)	$\Theta(n(n-k))$
Compute eigenvectors X of R	$\Theta(k^2)$
Compute eigenvectors $V = \hat{V}X$	$\Theta(nk^2)$

Table I
COST OF THE MERGE OPERATIONS

The two penultimate merges require $\frac{n^3}{4}$ operations and the rest of the algorithm only requires $\frac{n^3}{12}$ operations. Considering this result, computing both independent subproblems and merging steps in parallel seems to be important. It is worth noting that the greater the amount of deflation, the lesser the number of required operations, which leads to better performance. The amount of deflation depends on the eigenvalue distribution as well as the structure of the eigenvectors. In practice, most of the application matrices arising from engineering areas provide a reasonable amount of deflation, and so the D&C algorithm runs at less than $\mathcal{O}(n^{2.4})$ instead of $\mathcal{O}(n^3)$.

IV. PARALLELIZATION STRATEGY

PLASMA expresses algorithms as a sequential task flow. Dependencies between tasks are described through a set of qualifiers: INPUT, OUTPUT, and INOUT, which define the access type to the data. A master thread submits those tasks to the dynamic runtime QUARK [21]. This later analyzes the qualifiers to generate dependencies between tasks, and then schedule them in an out-of-order fashion over the working threads when their dependencies are satisfied. Algorithms exploiting this approach usually splits the work by (square or rectangular) tiles covering the matrix. The tiling allows the users to tune the execution from a few large efficient kernels to numerous smaller kernels that provide more parallelism and better scalability. The data is often distributed following the same cutting to improve data locality of the kernels.

Since computation of each eigenvector relies mostly on the secular equation (which computes the components v_i of the eigenvector) followed by the stabilization step which will also operates on the vector by itself, we decided to implement the D&C algorithm using a panel distribution of the tasks, thus keeping the LAPACK column major layout (rectangular tiles fashion). Operations on matrices are divided as a set of operations on eigenvectors panels. This greatly simplifies the management of the task dependencies in the divide and conquer process, and allows us to directly call LAPACK internal subroutines such as `LAED4` for computing the solution of the secular equation on each eigenvector. In contrast to the classical algorithm, the data dependencies tracking of the task flow implementation is not as straightforward and requires more attention because of the dynamic computation of the number of the non-deflated eigenvalues. Note that the required workspace of the algorithm depends on this computed deflation value and thus workspace sizes become dynamic if one wants to keep the extra space as low as possible. To overcome the dynamic output result, we decided to keep the same sequential task flow style which looks simple. However, in this case tasks related to the deflated and the non-deflated eigenvalues are submitted. This creates extra tasks without computational work. The overhead is considered marginal compared to the computational cost of other tasks. As a result, the generated task graph (DAG) is matrix independent: there will be as many tasks for a matrix with a lot of deflation as for a matrix without deflation.

Algorithm 1 Merge step

```
Compute deflation( V )                                  ▷ Deflate
for each eigenvectors panel p do
    PermuteV( V_p, V_d ) ▷ Permute and compress storage
    LAED4( V_p )                  ▷ Solve the secular equation
    ComputeLocalW( V_p, W_p )                ▷ Local reduction
end for
ReduceW( V, W )                                         ▷ Reduce
for each eigenvectors panel p do
    CopyBackDeflated( V_p, V_d )
    ComputeVect( V_p, W ) ▷ Stabilize new eigenvectors
    UpdateVect( V_p, V_d )                ▷ Update eigenvectors
end for
```

Most of the steps of the merging phase described in Section III can be redesigned in a parallel fashion by splitting the operations over a subset of eigenvectors, i.e., panel of the matrix V. The parallel algorithm is described in Algorithm 1 and is composed of eight tasks:

- *Compute deflation*: Find if there is a possible deflation, and generate the permutation to rearrange the eigenvalues/vectors in four groups: non-deflated of the first subproblem V_1, the correlated eigenvectors between V_1 and V_2, non-deflated of the second subproblem V_2, and the deflated eigenvalues. It returns the number of non-deflated eigenvalues.
- *PermuteV*: Rearranges a set of vectors from V, to a workspace in a compressed form: do not store the zeros below V_1, and above V_2 from Equation 6. Deflated vectors are sorted at the end of this buffer.
- *LAED4*: Solves the secular equation on each non-deflated eigenvalue computing the new eigenvalue λ of the rank-one modification problem R, and generating the components of the new eigenvector. Every task is a panel and thus works on a set of nb vectors.
- *ComputeLocalW*: The stabilization step described in Section III is split over two tasks. First is the parallel set of *ComputeLocalW* tasks which are independent for every panel, and then the *ReduceW* task perform a reduction to generate the stabilization vector W. Hence, the *ComputeLocalW* task consists of computing the

local contribution W_p to the stabilization vector W.

- *ReduceW*: Computes the stabilization vector W of all the panels.
- *CopyBackDeflated*: Copies back the deflated vectors to the end of father eigenvector matrix V. At this level, when the previous steps finished, we know all the permutation and sizes of deflation, so we can perform this copy back to allow using the workspace used for storing it in the next steps.
- *ComputeVect*: Stabilizes and compute the new non-deflated eigenvectors X. Thanks to the W vector.
- *UpdateVect*: Updates the non-deflated eigenvectors X by performing the matrix product $V = \tilde{V} \times X$ described in Equation 6.

As shown in Algorithm 1, only two kernels remain sequential and create a *join* operation: the *Compute deflation* and the *ReduceW*. However, these kernels are very fast (they constitute less than 1% of the overall computational time), and they are local to each merging phase. Therefore, two independent merging phase have two independent *Compute deflation* and *ReduceW* kernels that can run in parallel. This is not the case in the fork/join model provided by MKL LAPACK implementation that uses the multi-threaded BLAS library. A deep analysis of the task-flow model let us observe that some tasks have a dependency on every panel of the matrix resulting in about $\Theta(n)$ data tracking complexity. In order to minimize the amount of dependencies being tracked, we developed the GATHERV qualifier for the QUARK runtime. Therefore, all tasks have a constant number of dependencies. Those working on a panel have an extra dependency on the full matrix with the GATHERV flag, while the join task (*Compute deflation* and *ReduceW*) has a single INOUT dependency on the full matrix. All tasks with the GATHERV flag can be scheduled concurrently since the developer guarantees they work on independent data, and the join task waits for them to be completed before being scheduled. Then, it is possible to describe work similar to a fork/join model on each subproblem with a constant number of dependencies per task. This lowers the complexity of the dependency tracking algorithm. Furthermore, the task parallelism and granularity of this technique is controlled by the panel size nb which is a parameter that is defined by the problem size but also can be chosen by the user. As usual, nb has to be tunned to take advantage of criteria such as the number of cores (which define the amount of parallelism required to fulfill the cores) and the efficiency of the kernel itself (the computing kernel might have different behavior depending on small and large nb). Dependencies between the merging phases need to be handled correctly to allow independent subproblems to be solved in parallel. Since the tree is traversed in bottom-up fashion, different merging processes of the same level of the tree are always independent and thus can be solved in parallel. Moreover, we can state that the merging process between two different levels of the tree are independent if and only if they do not belong to the same branch of the tree. Figure 2 shows the task

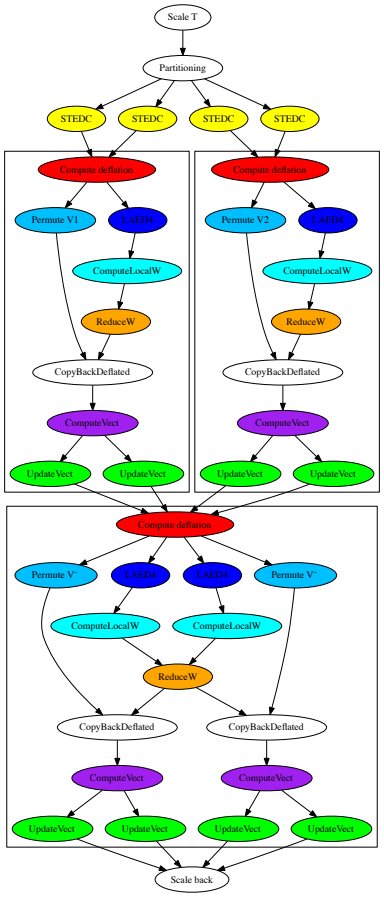


Figure 2. DAG of the D&C tridiagonal eigensolver for a matrix of size 1000, with a minimal partition size of 300, and a panel size of 500

Direct Acyclic Graph (DAG) and its dependencies obtained for a problem of size 1000, recursively partitioned down to achieve subproblem at the bottom of the tree of size less than or equal to 300 (minimal partition size is 300). This will result in four subproblems of size 250 each. We defined the panel size $nb = 500$ meaning that our tasks are split by panel of 500 columns. The first two nodes of the DAG are the tasks which prepare (scale if necessary the tridiagonal matrix T) and split the problem, then we can observe two sub-DAG (highlighted by the rectangular border) corresponding to the two independent merging processes; followed by the sub-DAG at the bottom that correspond to the final merge process. And finally if the original matrix T was scaled we need to apply the scale back. At the beginning, four independent eigenproblems are solved sequentially using the kernel *STEDC*. Then the runtime will schedule in parallel the two independent merging processes of a same level without

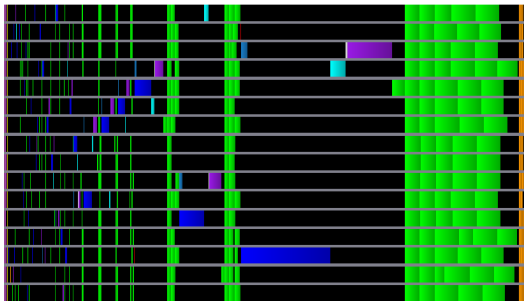
coordinating the progress of each merge. Figure 2 allows us to explain the importance of the parallelization we propose. Despite that the D&C algorithm exhibit tree parallelism, our proposition of breaking the panel into chunks of size nb will create extra tasks producing more parallelism that we can exploit either at the bottom or at the top of the DAG. Every level of the DAG consists of tasks that can run in parallel. For example, in Figure 2, for the first two merges, the fifth level of the DAG, assuming there is no deflation, each *LAED4* has a size of 500 (it corresponds to the merging process of the two sons of size 250 each). Since our panel $nb = 500$ we end up having one *LAED4* task for merging son_1 and son_2, and another parallel *LAED4* task for merging son_3 and son_4 while if we change our nb to be 100 we can create ten tasks. Note that we can observe in Figure 2 that the permute task can run in parallel with the *LAED4* task while in Algorithm 1 they are sequential. We analyzed the algorithm and found that we can also create extra parallel task by requiring extra workspace. For that we integrated a user option that allows the algorithm to use extra workspace and thus can create more parallel tasks. For example, the permutation can be executed in parallel with the *LAED4* if there is extra workspace. Similarly, the *CopyBack* can be executed in parallel with the *ComputeVect* if this extra workspace is allowed. In practice, the effect of this option can be seen on a machine with large number of cores where we need to exhibit parallelism as much as possible.

The execution traces of Figure 3 illustrates the different level of parallelization exploited to achieve efficient computation on multi-core and to avoid idle states. Those traces were obtained on a dual INTEL octo-core machine. The matrix used is of type 4 as described in Table III, and of size 10000×10000. We choose this type to illustrate the case with few deflated vectors implying expensive merge operations. The kernel's color and name are listed in Table II. Two additional tasks are present: *LASET* which initializes the workspace to identity matrix, and *SortEigenvectors* which sorts the eigenvectors by ascending order of the eigenvalues.

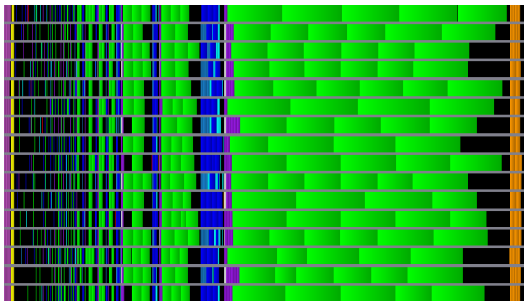
	UpdateVect		*ComputeVect*
	LAED4		*ComputeLocalW*
	SortEigenvectors		*STEDC*
	LASET		*Compute deflation*
	PermuteV		*CopyBackDeflated*

Table II
COLOR CODE OF THE KERNELS IN THE DAG AND EXECUTION TRACES

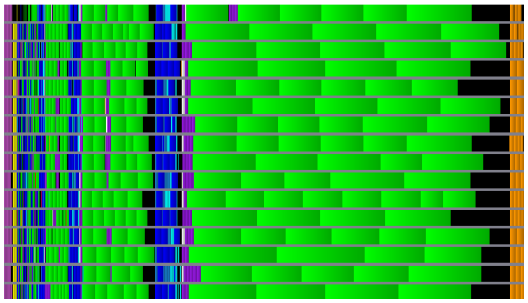
The sequential execution, which takes 18sec, shows that most of the time is spent on matrix products (*UpdateVect*). For this case, it represents around 90% of the overall time. The first idea is then to parallelize these operations (GEMM). The trace is illustrated in Figure 3(a) and it provides a makespan of only 4.3sec. This is similar to the INTEL MKL implementation on this architecture with a speedup of 4 with respect to the sequential implementation. In Figure 3(a) we can easily determine the level of the tree as well as each of the merging process. The evaluation of this trace in particular, the black color which corresponds to the idle time, make us notice that there is more room for improvements. Solving the secular equation (*LAED4* blue) represents around half of the execution time. It is then natural to parallelize this



(a) Multi-threaded vectors' update



(b) Multi-threaded merge operation



(c) Exploiting the independence between subproblems

Figure 3. Comparison of the execution traces on a type 4 matrix, with few deflated vectors, when adding optimizations on a 16 core architecture

operation, as well as the computation of the eigenvectors (purple). The result of this optimization is illustrated in Figure 3(b). It provides a speedup of a factor of two (1.8sec) on this 16 cores machine. Note that if the machine has more cores, we can expect higher speedup factor. The assessment of the trace shows that the last merging process at the top of the tree is the most time consuming (which confirm our complexity analysis) and could provide enough work for all the threads. However, some cores are still stalling during the small merges carried out at the bottom of the tree. Since these subproblems are independent, they can be solved in parallel to exploit the available resources. This last

improvement removes the synchronization barriers between the level of the tree, and reduces the idle time at the beginning of the trace. For instance, the two penultimate merges, previously computed one after the other, as shown in Figure 3(b), are now computed in parallel (Figure 3(c)). This provides a final improvement on the algorithm makespan, and a final speedup of twelve on this problem. These traces showed an example for a matrix without a lot of deflation. Since D&C behavior is matrix-dependent, we illustrated in Figure 4 the execution trace with a matrix of type 5 (see Table III) of size 10000×10000. As opposed to the previous example (type 4), this test case generates close to 100% of deflation. One can expect that the merging process is now summarized by the permutation kernels which mean that the computational bound operations (*UpdateVect* kernel), are mainly replaced by vector copies (Permute kernels) which are memory bound. Despite this constraint, one can observe, a good level of parallelism and small idle times with our proposed implementation. Note that due to the characteristics of the operations, the speedup expected will not be as high as the previous case.

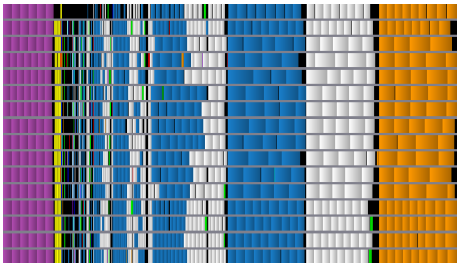


Figure 4. Execution trace on a type 5 matrix, with almost 100% deflation, on a 16 cores architecture

V. Performance and Numerical Accuracy Evaluation

Experiments were performed on a machine with 2 Intel Xeon E5-2650v2 CPUs running at 2.6GHz (16 cores total) and 64GB of memory. PLASMA was built using GCC 4.4.7, and linked with the sequential Intel MKL 11.2.0 and the HWLoc library. Experiments involving PLASMA were run with 16 threads bound to a dedicated core. Experiments with LAPACK and ScaLAPACK were using the Intel implementation, with 16 threads and 16 MPI processes respectively. Except for ScaLAPACK, `numactl --interleave=all` was used to dispatch the data in shared memory which will let the multi-thread MKL library performing better. For the comparison with MRRR, we used the MR^3-SMP 1.2, with 16 threads, built with the same GCC. The testing environment follows [5], [22], and is inspired from [23]. It uses a large spectrum of tridiagonal matrices in double precision described in Table III with varying sizes from 2500 to 25000. In our testing environment, the k parameter is arbitrarily set to $1.0e6$, and ulp is the relative unit-in-the-place, computed by the LAPACK subroutine `lamch` for precision.

Type	Description
1	$\lambda_1 = 1, \lambda_i = \frac{1}{k}, i = 2..n$
2	$\lambda_i = 1, i = 1..n-1, \lambda_n = \frac{1}{k}$
3	$\lambda_i = k^{-\frac{i-1}{n-1}}, i = 1..n$
4	$\lambda_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{k}), i = 1..n$
5	n random numbers with $log(i)$ uniformly distributed
6	n random numbers
7	$\lambda_i = ulp * i, i = 1..n-1, \lambda_n = 1$
8	$\lambda_1 = ulp, \lambda_i = 1 + i \times \sqrt{ulp}, i = 2..n-1, \lambda_n = 2$
9	$\lambda_1 = 1, \lambda_i = \lambda_{i-1} + 100 * ulp, i = 2..n$
10	(1,2,1) tridiagonal matrix
11	Wilkinson matrix
12	Clement matrix
13	Legendre matrix
14	Laguerre matrix
15	Hermite matrix

Table III
DESCRIPTION OF THE TEST MATRICES

For scalability and comparison with respect to the Intel MKL library implementation, we used matrices of type 2, 3, and 4, because they present different properties, with about 20%, 50% and 100% of deflation respectively. Our experiments showed that the proposed set of experiments are representatives for the matrices presented in Table III. Figure 5 presents the scalability of our implementation from 1 to 16 threads. Small amount of deflation cases showed up to $12\times$ speedup using the 16 threads. This behavior is similar to the one illustrated in Section IV for the execution traces. When the deflation ratio is high, the speedup decreases and highlights the transition from a compute bound to a memory bound algorithm. For memory bound operations (Permute kernels) the speedup can be determined by the bandwidth. We can easily see that 4 threads are able to saturate the bandwidth of the first socket and thus the speedup stagnate around 4 till we start using the second socket (>8 threads).

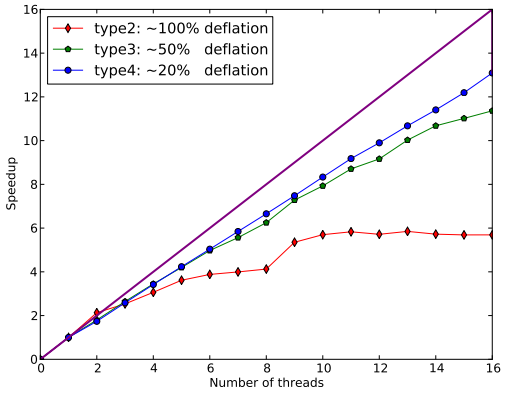


Figure 5. Scalability of the D&C algorithm on type 2, 3 and 4 matrices

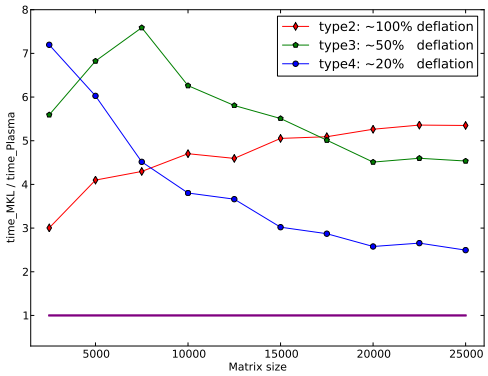


Figure 6. Speedup of D&C algorithm with respect to the INTEL MKL LAPACK implementation

Figure 6 presents the speedup of our algorithm with respect to the INTEL MKL implementation. When the deflation level is large, our algorithm takes advantage of the parallelization of the subproblems and the secular equation. As a consequence, it is four to six times faster. When the deflation level is small, and when the machine has small number of cores, the cost of the algorithm is mainly bound by the cost of GEMM operations. Thus, one can expect that the model based on multi-threaded BLAS (the LAPACK routines in the INTEL MKL library) will marginally decrease the speedup for large matrix sizes.

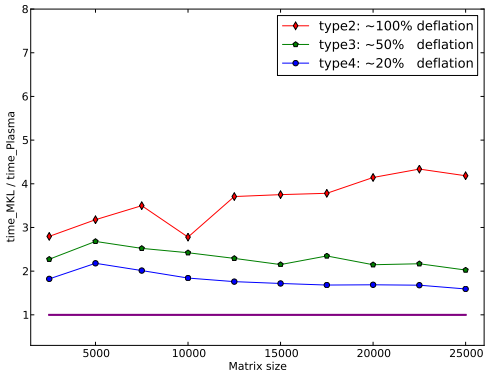


Figure 7. Speedup of D&C algorithm with respect to the INTEL MKL SCALAPACK implementation

Figure 7 presents the speedup with respect to the INTEL MKL SCALAPACK implementation. Contrary to LAPACK, SCALAPACK already solves independent subproblems in parallel. Furthermore, the programming models are different. As SCALAPACK is developed for distributed architectures, the memory is naturally distributed among the different NUMA nodes before calling the tridiagonal eigensolver. In PLASMA, we use `numactl --interleave=all` to distribute the data among different NUMA nodes. However, with this approach the data is not as close to the compute node as in SCALAPACK. On the other side, SCALAPACK prevents direct access to the data and data copies are required for exchanges between NUMA nodes. The speedup is then not as good as the one shown above, but remains around two for matrices with more than 20% of deflation. It also reaches four for test cases with almost 100% deflation.

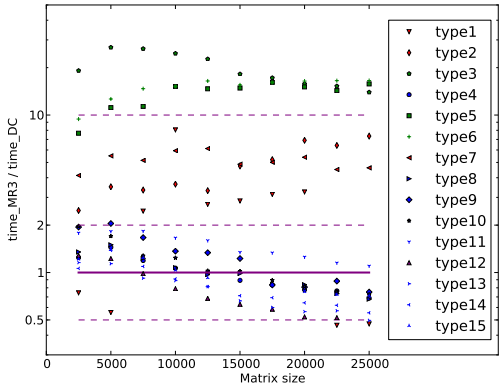
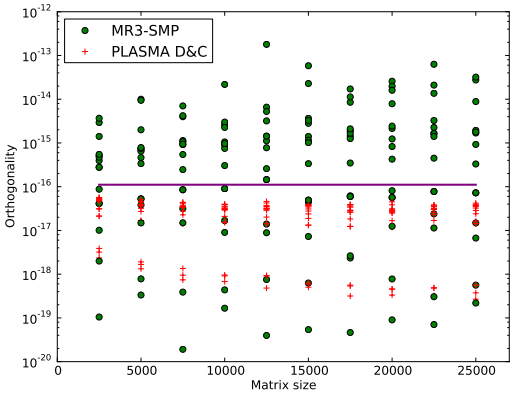


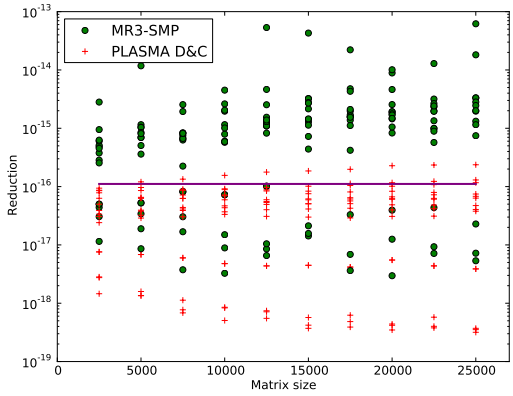
Figure 8. Timing of the MR^3-SMP implementation versus PLASMA D&C

The comparison with MRRR is more complex than the comparison with other D&C eigensolvers. Indeed, D&C is efficient when eigenvalues are clustered, whereas MRRR is fast when the eigenvalues are well separated. D&C requires larger workspace, but its accuracy is better. All those parameters suggest that choosing one eigensolver is problem-dependent. As reported in Figure 8, the timing comparison with respect to MR^3-SMP is matrix-dependent. Both implementations are always faster than the corresponding INTEL MKL implementations, with the same accuracy. The slower MR^3-SMP is, the faster D&C becomes, and, conversely, as stated by the theory. For most of the test cases, our new algorithm is faster than MR^3-SMP and can be up to $25\times$ faster except for some cases where it can be at max $2\times$ slower. Considering the improvement with respect to existing D&C implementations, those results are interesting, because a larger set of matrices become faster to solve using D&C than using MRRR. On an application where performance is the main asset, we can suppose that using D&C or MRRR will depend on the matrix used. However, the main asset of D&C is the accuracy provided, and it is better than the obtained accuracy with MRRR on both the orthogonality of the eigenvectors and the reduction of the tridiagonal matrix. In theory, for a matrix of size n and a machine precision of ϵ, D&C achieves errors of size $O(\sqrt{n}\epsilon)$, whereas MRRR error is in $O(n\epsilon)$.

Figures 9(a) and 9(b) present the eigenvectors orthogonality, and the reduction of the tridiagonal matrix precision, respectively. Results shows that the precision of our implementation is comparable to that of D&C LAPACK, and MR^3-SMP achieves the same results as MRRR LAPACK. As predicted by the theory, D&C implementations are always more accurate than MRRR's, with a difference



(a) Orthogonality: $\frac{||I-QQ^T||}{N}$



(b) Reduction: $\frac{||T-Q\Sigma Q^T||}{||T||\times N}$

Figure 9. Stability of both MR^3-SMP and PLASMA D&C on the full set of matrices with different sizes

between one and two digits in our experiments in favor of D&C. In addition, the worst accuracy case for MR^3-SMP is the fastest run: when MR^3-SMP is faster than PLASMA, the achieved accuracy is often less than the machine precision. MRRR can achieve good accuracy, because all eigenvalues are computed before the eigenvectors. This provides additional information on the matrix properties to stabilize the process. As the tridiagonal eigensolver is the critical part of the complete symmetric eigensolver in terms of accuracy, our results (always close to the machine precision) are interesting, because they are as accurate as the corresponding reduction stage, and they show that multiple threads do not degrade the results. Previous implementations exhibit the same accuracy, but the set of matrices where D&C outperformed MRRR was smaller. Figure 10 shows the timing of the MR^3-SMP and the PLASMA D&C algorithm on a set of application matrices from the LAPACK `stetester` [1] and described in [22]. Our D&C implementation outperforms MR^3-SMP on almost all cases while providing a better

[1] http://crd-legacy.lbl.gov/~osni/Codes/stetester/DATA

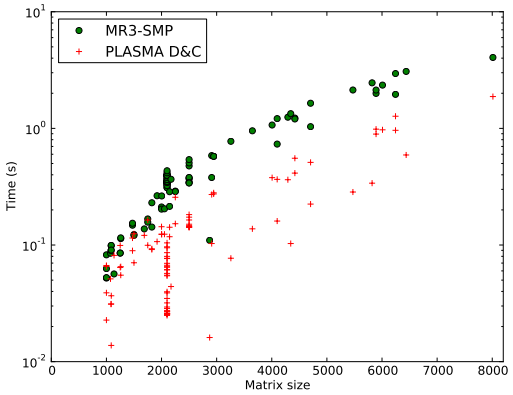


Figure 10. Application matrices

accuracy. However, those matrices are small and it is hard to conclude anything about scalability and efficiency.

VI. CONCLUSION

We presented a new Divide and Conquer implementation, exploiting parallelism through a task-flow model. According to the experiments we conducted, our implementation outperforms existing D&C implementations with the same accuracy. The task-based approach, using the runtime QUARK to schedule tasks, provides good performance, as seen previously with the implementation of the reduction to tridiagonal form. In addition, performance experiments showed that the implementation is competitive with the best MRRR implementation on shared-memory architectures. Considering the assets and the drawbacks of both algorithms, and the fact that the problem is mainly matrix-dependent, choosing one eigensolver depends on the application. The extra amount of memory required by D&C could be problematic, but the robustness of the algorithm ensures that we obtain an accurate solution. In addition, our algorithm presents a speedup on matrices extracted from real-life applications. Those matrices are well-known, and we suppose they represent a good set to demonstrate the benefit of our new Divide and Conquer implementation.

Our main improvement was to express more parallelism during the merge step where the quadratic operations become costly as long as the cubic operations are well parallelized. A recent study by Li [24] showed that the matrix products could be improved with the use of hierarchical matrices, reducing the cubic part with the same accuracy. Combining both solutions should provide a fast and accurate solution, while reducing the memory space required.

For future work, we plan to study the implementation for both heterogeneous and distributed architectures, in the MAGMA and DPLASMA libraries. As the Singular Value Decomposition (SVD) follows the same scheme as the symmetric eigenproblem, by reducing the initial matrix to bidiagonal form and using a Divide and Conquer algorithm

as bidiagonal solver, it is also a good candidate for applying the ideas of this paper. bidiagonal solver.

ACKNOWLEDGMENTS

This material is based upon work supported by the Inria associate team MORSE and by the National Science Foundation under Grant No. ACI-1339822 and the Department of Energy.

REFERENCES

[1] J. Dongarra, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, A. YarKhan, W. Alvaro, M. Faverge, A. Haidar, J. Hoffman, E. Agullo, A. Buttari, and B. Hadri, "Plasma users' guide: Parallel linear algebra software for multicore architectures," 2010.

[2] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra, "Multithreading in the plasma library," 2013.

[3] A. Haidar, H. Ltaief, and J. Dongarra, "Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, pp. 8:1–8:11.

[4] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[5] J. Demmel, O. Marques, B. N. Parlett, and C. Vomel, "Performance and accuracy of lapack's symmetric tridiagonal eigensolvers." *SIAM J. Scientific Computing*, vol. 30, no. 3, pp. 1508–1526, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/siamsc/siamsc30.html#DemmelMPV08

[6] T. Auckenthaler, V. Blum, H. J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations," *Parallel Comput.*, vol. 37, no. 12, pp. 783–794, Dec. 2011.

[7] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: SIAM, 1992.

[8] Y. Nakatsukasa and N. J. Higham, "Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the svd," *SIAM J. Scientific Computing*, vol. 35, pp. A1325-A1349, 2013.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997.

[10] J. D. Rutter, "A serial implementation of cuppen's divide and conquer algorithm for the symmetric eigenvalue problem," LAPACK Working Note, Tech. Rep. 69, Mar. 1994. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn69.pdf

[11] J. Dongarra and D. C. Sorensen, "A fully parallel algorithm for the symmetric eigenvalue problem." in *PPSC*, C. W. Gear and R. G. Voigt, Eds. SIAM, pp. 139–154.

[12] F. Tisseur and J. Dongarra, "Parallelizing the divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem on distributed memory architectures," *SIAM J. SCI. COMPUT*, vol. 20, pp. 2223–2236, 1998.

[13] Intel, "Math kernel library," https://software.intel.com/.

[14] I. S. Dhillon, "A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem," Ph.D. dissertation, EECS Department, University of California, Berkeley, Oct 1997. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/5888.html

[15] M. Petschow and P. Bientinesi, "MR^3-SMP: A symmetric tridiagonal eigensolver for multi-core architectures," *Parallel Computing*, vol. 37, no. 12, 2011.

[16] D. Liu, R. Li, D. J. Lilja, and W. Xiao, "A divide-and-conquer approach for solving singular value decomposition on a heterogeneous system," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/2482767.2482813

[17] J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem," vol. 36, no. 2, pp. 177–195, Apr. 1981.

[18] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem," *SIAM J. Matrix Anal. Appl.*, vol. 16, no. 1, pp. 172–191, Jan. 1995.

[19] G. H. Golub, "Some modified matrix eigenvalue problems," *SIAM Review*, vol. 15, no. 2, pp. 318–334, Apr. 1973.

[20] P. Arbenz and G. H. Golub, "On the spectral decomposition of hermitian matrices modified by low rank perturbations," *SIAM J. Matrix Anal. Appl.*, vol. 9, no. 1, pp. 40–58, 1988.

[21] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2011.

[22] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vomel, "A testing infrastructure for LAPACK's symmetric eigensolvers," LAPACK Working Note, Tech. Rep. 182, 2007.

[23] A. Haidar, H. Ltaief, and J. Dongarra, "Toward a high performance tile divide and conquer algorithm for the dense symmetric eigenvalue problem," *SIAM J. Scientific Computing*, vol. 34, pp. C249-C274, 2012.

[24] S. Li, M. Gu, L. Cheng, X. Chi, and M. Sun, "An accelerated divide-and-conquer algorithm for the bidiagonal SVD problem," vol. 35, no. 3, pp. 1038–1057, 2014.